intel

APPLICATION NOTE

AP-222

November 1984

Asynchronous and SDLC Communications with 82530

SHARAD GANDHI

Order Number: 231262-001

ASYNCHRONOUS AND SDLC COMMUNICATIONS WITH 82530

CONTENTS

INTRODUCTION

INTEL's 82530, Serial Communications Controller (SCC), is a dual channel, multi-protocol data communications peripheral. It is designed to interface to high speed communications lines using asynchronous, byte synchronous and bit synchronous protocols. It runs up to 1.5 Mbits/sec, has on-chip baud rate generators and on-chip NRZI encoding and decoding circuits — very useful for SDLC communication. This application note shows how to write I/O drivers for the 82530 to do initialization and data links using asynchronous (ASYNC) and SDLC protocols. The appendix includes sections to show how the on-chip baud rate generators could be programmed, how the modem control pins could be used and how the 82530 could be interfaced to INTEL's 80186/188 processors.

This article deals with the software for the following:

1. SCC port definition
2. Accessing the SCC registers
3. Initialization for ASYNC communication
4. ASYNC communication in polling mode
5. ASYNC communication in interrupt mode
6. Initialization for SDLC communication
7. SDLC frame reception
8. SDLC frame transmission
9. SDLC interrupt routines

The description is written around illustrations of the actual software written in PLM86 for a 80186 - 82530 system.

I. SCC Port Definition

The Figure 1 shows how the 4 ports (2 per channel) of the SCC can be defined. Note that the sequence of ports in the ascending order of addresses is *not* the one that is normally expected. In the ascending order it is: command (B), data (B), command (A) and data (A). In an 80186 - 82530 system, the interconnection is as follows:

	PCSn	---	CS	
	A1	---	D/C	
80186 pins	A2	---	A/B	82530 pins
	RD	---	RD	
	WR	---	WR	

2. Accessing the SCC Registers

The SCC has 16 registers on each of the channels (A and B). For each channel there is only one port, the command port, to access all the registers. The register #0 can be always accessed directly through the command port. All other registers are accessed indirectly through register #0. First, the number of the register to be accessed is written to the register #0 - see the statement, in Figure 2: 'output (ch_a_command) = reg_no and 0fh'. Then, the desired register is written to or read out of the register #0. The Figure 2 shows 4 procedures: rra and wra, for reading and writing channel A registers; rrb and wrb, for reading and writing channel B registers. The read procedures are of the type 'byte' - they return the contents of the register being read. The write procedures require two parameters - the register number and the value to be written.

```
/*-------------------------------------------------------------------------*/

declare ch_b_command  literally 'pcs5 + 0', /* scc channel_b command word*/
        ch_b_data     literally 'pcs5 + 2', /* scc channel_b data word */
        ch_a_command  literally 'pcs5 + 4', /* scc channel_a command word */
        ch_a_data     literally 'pcs5 + 6'; /* scc channel_a data word */

/*-------------------------------------------------------------------------*/
                                                              231262-1
```

Figure 1. SCC Port Definition

```
/*-------------------------------------------------------------------------*/

/* read selected scc register */

rra: procedure (reg_no) byte;
     declare reg_no byte;

     if (reg_no and 0fh) <> 0
     then output(ch_a_command) = reg_no and 0fh;
     return input(ch_a_command);
end rra;

rrb: procedure (reg_no) byte;
     declare reg_no byte;

     if (reg_no and 0fh) <> 0
     then output (ch_b_command) = reg_no and 0fh;
     return input(ch_b_command);
end rrb;


/* write selected scc register */

wra: procedure (reg_no, value);
     declare reg_no byte;
     declare value byte;

     if (reg_no and 0fh) <> 0
     then output (ch_a_command) = reg_no and 0fh;
     output (ch_a_command) = value;
end wra;

wrb: procedure (reg_no, value);
     declare reg_no byte;
     declare value byte;

     if (reg_no and 0fh) <> 0
     then output (ch_b_command) = reg_no and 0fh;
     output (ch_b_command) = value;
end wrb;

/*-------------------------------------------------------------------------*/
```

231262-2

Figure 2. Accessing the SCC Registers

3. Initialization for ASYNC Operation

Channel B of the SCC is used to perform ASYNC communication. Figure 3 shows how the channel B is initialized and configured for ASYNC operation. This is done by writing the various channel B registers with the proper parameters as shown. The comments in the program show what is achieved by each statement. After a software reset of the channel, register #4 should be written before writing to the other registers. The on-chip Baud Rate Generator is used to generate a 1200 bits/sec clock for both the transmitter and the receiver. The interrupts for transmitter and/or receiver are enabled only for the interrupt mode of operation; for polling, interrupts must be kept disabled.

4. ASYNC Communication in Polling Mode

Figure 4 shows the procedures for reading in a received character from the 82530 (scc__in) and for writing out a character to the 82530 (scc__out) in the polling mode.

The scc__in procedure returns a byte value which is the character read in. The receiver is polled to find if a character has been received by the SCC. Only when a character has been received, the character is read in from the data port of the SCC channel B.

The scc__out procedure requires a byte parameter which is the character being written out. The transmit-

```
/*-------------------------------------------------------------------------*/

scc_init_b: procedure;

/* scc ch B register initialization for ASYNC mode */

     call wrb(09, 01000000b);     /* channel B reset */
     call wrb(04, 11001110b);     /* 2 stop, no parity, brf = 64x */
     call wrb(02, 00100000b);     /* vector = 20h */
     call wrb(03, 11000000b);     /* rx 8 bits/char, no auto-enable */
     call wrb(05, 01100000b);     /* tx 8 bits/char */
     call wrb(06, 00000000b);
     call wrb(07, 00000000b);
     call wrb(09, 00000001b);     /* vector includes status */
     call wrb(10, 00000000b);
     call wrb(11, 01010110b);     /* rxc = txc = BRG , trxc = BRG out */
     call wrb(12, 00011000b);     /* to generate 1200 baud, x64 @ 4 mhz */
     call wrb(13, 00000000b);
     call wrb(14, 00000011b);     /* BRG source = SYS CLK, enable BRG */
     call wrb(15, 00000000b);     /* all ext status interrupts off */

    /* enables */

     call wrb(03, 11000001b);      /* scc-b receive enable */
     call wrb(05, 11101010b);      /* scc-b transmit enable, dtr on, rts on */

    /* enable interrupts - only for interrupt driven ASYNC I/O */

     call wrb(09, 00001001b);     /* master IE, vector includes status */
     call wrb(01, 00010011b);     /* tx ,rx, ext interrupts enable */

end scc_init_b;

/*-------------------------------------------------------------------------*/
                                                                  231262-3
```

Figure 3. Initialization for ASYNC Communication

```
/*-------------------------------------------------------------------------*/

/* scc data character input from channel B */

scc_in: procedure byte;

     declare char byte;

     do while (input(ch_b_command) and 1h) = 0; end;
     char = input(ch_b_data);    /* if rx data character is available */
     return char;                /* then input it to buffer */

end scc_in;

/* scc data character output to channel B */

scc_out: procedure (char);

     declare char byte;

     do while (input(ch_b_command) and 4h) = 0; end;
     output(ch_b_data) = char;   /* if tx buff empty then transfer the */
                                 /* data character to tx buff */

end scc_out;

/*-------------------------------------------------------------------------*/
```

231262-4

Figure 4. ASYNC Communication in Polling Mode

ter is polled for being ready to transmit the next character before writing the character out to the data port of SCC channel B.

Typical calls to these procedures are:

```
abc_variable = scc_in;
call scc_out (xyz_variable);
```

5. ASYNC Communication in Interrupt Mode

In contrast to polling for the receiver and/or the transmitter to be ready with/for the next character, the 82530 can be made to interrupt when it is ready to do receive or transmit.

The on-chip interrupt controller of the SCC can be made to operate in the vectored mode. In this mode, it generates interrupt vectors that are characteristic of the event causing the interrupt. For the example here, the vector base is programmed at 20h and 'Vector Includes Status' (VIS) mode is set - WR9 = XXX0XX01. Vectors and the associated events are:

Vector	Procedure	Event Causing Interrupt
20h	txintr_b	ch_b - transmit buffer empty
22h	esi_b	ch_b - external/status change
24h	rxintr_b	ch_b - receive character available
26h	src_b	ch_b - special receive condition
28h	txintr_a	ch_a - transmit buffer empty
2ah	esi_a	ch_a - external/status change
2ch	rxintr_a	ch_a - receive character available
2eh	src_a	ch_a - special receive condition

NOTE:
Odd vector numbers do not exist.

Figure 5 shows the interrupt procedures for the channel B operating in ASYNC mode. The transmitter buffer empty interrupt occurs when the transmitter can accept one more character to output. In the interrupt procedure for transmit, the byte char_out_530 is output. Following this, is an epiloge that is *common to all the*

interrupt procedures; the first statement is an end of interrupt command to the 82530 - *note* that it is issued to *channel A* - and the second is an End of Interrupt (EOI) command to the 80186 interrupt controller which is, in fact, receiving the interrupt from the 82530.

The receive buffer full interrupt occurs when the receiver has at least one character in its buffer, waiting to be read in by the CPU.

The esi_b is not enabled to occur and src_b cannot occur in the ASYNC mode unless the receiver is overrun or a parity error occurs.

```
/*-------------------------------------------------------------------------*/

/* channel B interrupt procedures */

txintr_b:    procedure    interrupt 20h;

    output (ch_b_data) = char_out_530;

    call wra(00,38h);                /* reset highest IUS */
    output (eoir_186)  = 8000h;      /* non specific EOI */
    return;
end txintr_b;

esi_b:       procedure    interrupt 22h;

    call wrb(00,10h);                /* reset ESI */

    call wra(00,38h);                /* reset highest IUS */
    output (eoir_186)  = 8000h;      /* non specific EOI */
    return;
end esi_b;

rxintr_b:    procedure    interrupt 24h;

    char_in_530 = input (ch_b_data);

    call wra(00,38h);                /* reset highest IUS */
    output (eoir_186)  = 8000h;      /* non specific EOI */
    return;
end rxintr_b;

src_b:       procedure    interrupt 26h;

    call wrb(00,30h);                /* error reset */

    call wra(00,38h);                /* reset highest IUS */
    output (eoir_186)  = 8000h;      /* non specific EOI */
    return;
end src_b;

/*-------------------------------------------------------------------------*/
```

231262-5

Figure 5. ASYNC Communication in Interrupt Mode

6. Initialization for SDLC Communication

Channel A of the SCC is programmed for being used for SDLC operation. It uses the DMA channels on the 80186. Figure 6 shows the initialization procedure for channel A. The comments in the software show the effect of each statement. The on-chip Baud Rate Generator is used to generate a clock of 125 KHz both for reception and transmission. This procedure is just to prepare the channel A for SDLC operation. The actual transmission and reception of frames is done using the procedures described further.

7. SDLC Frame Reception

Figure 7 shows the entire set-up necessary to receive a SDLC frame. First the DMA controller is programmed with the receive buffer address (@rx__buff), byte count, mode etc and is also enabled. Then a flag indicating reception of the frame is reset. An Error Reset command is issued to clear up any pending error conditions. The receive interrupt is enabled to occur at the end of frame reception (Special Receive Condition); lastly, the receiver is enabled and put in the Hunt mode (to detect the SDLC flag). When the first flag is detected on the RxDA pin, it goes from the Hunt to the Sync mode. It receives the frame and the end of frame interrupt (src__b, vector = 2eh) occurs.

8. SDLC Frame Transmission

Figure 8 shows the procedure for transmitting a SDLC frame once the channel A is initialized. The DMA controller is initialized with the transmit buffer address (@tx__buff (1)) - *note*, it is the second byte of the transmit buffer - and the byte count - again one less than the total buffer length. This is done because the first byte in the buffer is output directly using an I/O instruction and not by DMA. Then the flag indicating frame transmitted is reset. The events following are very critical in sequence:

a. Reset external status interrupts
b. Enable the transmitter
c. Reset transmit CRC
d. Enable transmitter underrun interrupt
e. Enable the DMA controller
f. Output first byte of the transmit block to data port
g. Reset Transmit Underrun Latch

```
/*-------------------------------------------------------------------------*/

scc_init_a: procedure;

/* scc ch A register initialization for SDLC mode */

      call wra(09, 10000000b);     /* channel A reset */
      call wra(04, 00100000b);     /* SDLC mode */
      call wra(01, 01100000b);     /* DMA for Rx */
      call wra(03, 11000000b);     /* 8 bit Rx char, Rx disable */
      call wra(05, 01100000b);     /* 8 bit Tx char, Tx disable */
      call wra(06, 01010101b);     /* node address */
      call wra(07, 01111110b);     /* SDLC flag */
      call wra(10, 10000000b);     /* preset CRC, NRZ encoding */
      call wra(11, 01010110b);     /* rxc = txc = BRG , trxc = BRG out */
      call wra(12, 00001110b);     /* to generate 125 Kbaud, x1 @ 4 mhz */
      call wra(13, 00000000b);
      call wra(14, 00000110b);     /* BRG source = SYS CLK, DMA for Tx */
      call wra(15, 00000000b);     /* all ext status interrupts off */

    /* enables */

      call wra(14, 00000111b);      /* enable : BRG */
      call wra(01, 11100000b);      /* enable : dreq */
      call wra(09, 00001001b);      /* master IE, vector includes status */

end scc_init_a;

/*-------------------------------------------------------------------------*/
```

231262-6

Figure 6. Initialization for SDLC Communication

```
/*-------------------------------------------------------------------------*/

rx_init: procedure;

     declare dma_0_mode literally '1010001001000000b';
     /* src=IO, dest=M(inc), sync=src, TC, noint, priority, byte */

     outword(dma_0_dpl) = low16(@rx_buff);
     outword(dma_0_dph) = high16(@rx_buff);
     outword(dma_0_spl) = ch_a_data;
     outword(dma_0_sph) = 0;
     outword(dma_0_tc)  = block_length + 2;        /* +2 for CRC */
     outword(dma_0_cw)  = dma_0_mode or 0006h;     /* start DMA channel 0 */

     frame_recd = 0;                   /* reset frame received flag */

     call wra(00,30h);                 /* error reset */
     call wra(01, 11111001b);          /* sp. cond intr only, ext int enable */
     call wra(03, 11010001b);          /* enable receiver, enter hunt mode */

end rx_init;

/*-------------------------------------------------------------------------*/
                                                              231262-7
```

Figure 7. SDLC Frame Reception

```
/*-------------------------------------------------------------------------*/

tx_init: procedure;

     declare dma_1_mode literally '0001011010000000b';
     /* src=M(inc), dest=IO, sync=dest, TC, noint, noprior, byte */

     outword(dma_1_spl) = low16(@tx_buff(1));
     outword(dma_1_sph) = high16(@tx_buff(1));
     outword(dma_1_dpl) = ch_a_data;
     outword(dma_1_dph) = 0;
     outword(dma_1_tc)  = block_length - 1;     /* -1 for first byte */

     frame_tx = 0;                     /* reset frame transmitted flag */

     call wra(00, 00010000b);          /* reset ESI */
     call wra(05, 01101011b);          /* enable transmitter */
     call wra(00, 10101000b);          /* reset tx CRC, TxINT pending */
     call wra(15, 01000000b);          /* enable : TxU int */

     outword(dma_1_cw)  = dma_1_mode or 0006h;     /* start DMA channel 1 */
     output(ch_a_data) = tx_buff(0);     /* first byte - address field */
     call wra(00, 11000000b);            /* Reset Tx Underrun latch */

end tx_init;

/*-------------------------------------------------------------------------*/
                                                              231262-8
```

Figure 8. SDLC Frame Transmission

```
/*-------------------------------------------------------------------------*/

/* channel A interrupt procedures */

txintr_a:     procedure     interrupt 28h;

    call wra(00,38h);                   /* reset highest IUS */
    output (eoir_186)  = 8000h;         /* non specific EOI */
    return;
end txintr_a;

esi_a:        procedure     interrupt 2ah;

    call wra(00,10h);                   /* reset ESI */
    tx_stat = rra(0);                   /* read in status */
    frame_tx = 0ffh;                    /* set frame transmitted flag */

    call wra(00,38h);                   /* reset highest IUS */
    output (eoir_186)  = 8000h;         /* non specific EOI */
    return;
end esi_a;

rxintr_a:     procedure     interrupt 2ch;

    call wra(00,38h);                   /* reset highest IUS */
    output (eoir_186)  = 8000h;         /* non specific EOI */
    return;
end rxintr_a;

src_a:        procedure     interrupt 2eh;

    rx_stat = rra(1);
    call wra(00,30h);                   /* error reset */
    call wra(03,11000000b);             /* disable rx */
    frame_recd = 0ffh;                  /* set frame received flag */

    call wra(00,38h);                   /* reset highest IUS */
    output (eoir_186)  = 8000h;         /* non specific EOI */
    return;
end src_a;

/*-------------------------------------------------------------------------*/
```

231262-9

Figure 9. SDLC Interrupt Routines

The frame gets transmitted out with all bytes, except the first one, being fetched by the SCC using the DMA controller. At the end of the block the DMA controller stops supplying bytes to the SCC. This makes the transmitter underrun. Since the Transmitter Underrun Latch is in the reset state at this moment, the CRC bytes are appended by the SCC at the end of the transmit block going out. An External Status Change interrupt (esi__a, vector = 2ah) is generated with the bit for transmitter underrun set in RR0 register. This interrupt occurs when the CRC is being transmitted out and *not* when the frame is completely transmitted out.

9. SDLC Interrupt Routines

Figure 9 shows all the interrupt procedures for channel A when operating in the SDLC mode. The procedures of significance here are esi__a and src__a.

The end of frame reception results in the src__a procedure getting executed. Here the status in register RR1 is stored in a variable rx__stat for future examination. Any error bits set in status are reset, receiver is disabled and the flag indicating reception of a new frame is set.

The esi__a procedure is executed when CRC of the transmitted frame is just going out of the SCC. Reset External Status Interrupt command is executed, the external status is stored in a variable tx__stat for future examination and the flag indicating transmission of the frame is set.

End of frame processing is required after both of these interrupt procedures. It involves looking at rx__stat and tx__stat and checking if the desired operation was successful. The buffers used, may have to be recovered or new ones obtained to start another frame transmission or reception.

CONCLUSIONS

This article should ease the process of writing a complete data link driver for ASYNC and SDLC modes since most of the hardware dependent procedures are illustrated here. It was a conscious decision to make the procedures as small and easy to understand as possible. This had to be done at the expense of making the procedures general and not dealing with various exception conditions that can occur.

REFERENCES

1. 82530 Data Sheet, Order #230834-001
2. 82530 SCC Technical Manual, Order #230925-001

APPENDIX A
82530—BAUD RATE GENERATORS

The 82530 has two Baud Rate Generators (BRG) on chip—one for each channel. They are used to provide the baud rate or serial clock for receive and transmit operations. This article describes how the BRG can be programmed and used.

The BRG for each channel is totally independent of each other and have to be programmed separately for each channel. This article describes how any one of the two BRGs can be programmed for operation. To use the BRG, four steps have to be performed:

1. Determine the Baud Rate Time Constant (BRTC) to be programmed into registers WR12 (LSB) and WR13 (MSB).
2. Program in register WR11, to specify where the output of the BRG must go to.
3. Program the clock source to the BRG in register WR14.
4. Enable the BRG.

Step 1: Baud Rate Time Constant (BRTC)

The BRTC is determined by a simple formula:

$$\text{BRTC} = \frac{\text{Serial Clock Frequency}}{2 \times (\text{Baud Rate} \times \text{Baud Rate Factor})} - 2$$

Example:

For Serial Clock Frequency = 4 MHz

Baud Rate = 9600

Baud Rate Factor = 16

$$\text{BRTC} = \frac{4000000}{2 \times (9600 \times 16)} - 2$$

$$= 13.021 - 2 = 11.021$$

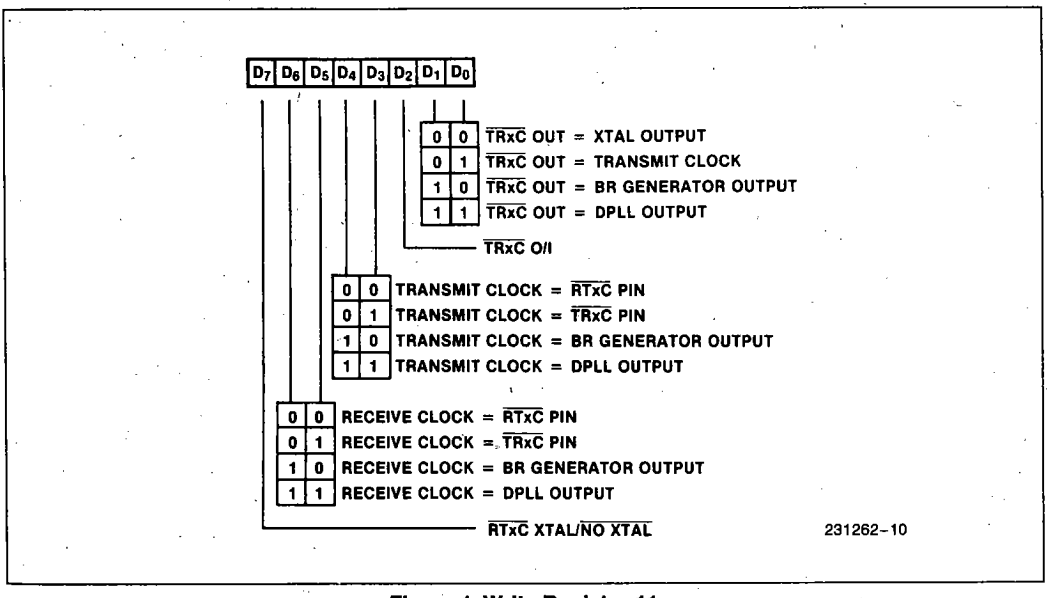


Figure 1. Write Register 11

Table 1. BRTC - Baud Rate Time Constant

		Baud Rate Factor			
		1	16	32	64
Baud Rate	9600	206.333	11.021	4.510	1.255
	4800	414.667	24.042	11.021	4.510
	2400	831.333	50.083	24.042	11.021
	1200	1664.667	102.167	50.083	24.042
	600	3331.333	206.333	102.167	50.083
	300	6664.667	414.667	206.333	102.167

Since only integers can be written into the registers WR12/WR13 this will have to be rounded off to 11 and it will result in an error of:

$$\frac{\text{fraction}}{\text{BRTC}} \times 100 = \frac{0.021}{11.021} \times 100 = 0.19\%$$

This error indicates that the baud rate signal generated by the BRG does not provide the exact frequency required by the system. This error is more serious for smaller baud rate factors. For asynchronous systems, errors up to 5% are considered acceptable.

Note that for BRTC = 0, BRG output frequency = 1/4 × Serial Clock Freq.

Table 1 shows the BRTC for a 4 MHz serial clock with various baud rates on the Y - axis and baud rate factors on the X - axis. The constant that is really programmed into registers WR12/WR13 is the integer closest to the BRTC value shown in the table.

Step 2: BRG Output

The output of the BRG can be directed to the Receiver, Transmitter and the TRxC output. This is programmed by setting bits D6 D5, bits D4 D3, and bits D1 D0 in register WR11 to 10. See Figure 1. The output of the BRG can also be directed to the Digital Phase Locked Loop (DPLL) for the on-chip decoding of the NRZI encoded received data signal. This is done by writing 100 into bits D7 D6 D5 of register WR14 as shown in Figure 2.

Step 3: BRG Source Clock

Register WR14 is used to select the input clock to the BRG. See Figure 2.

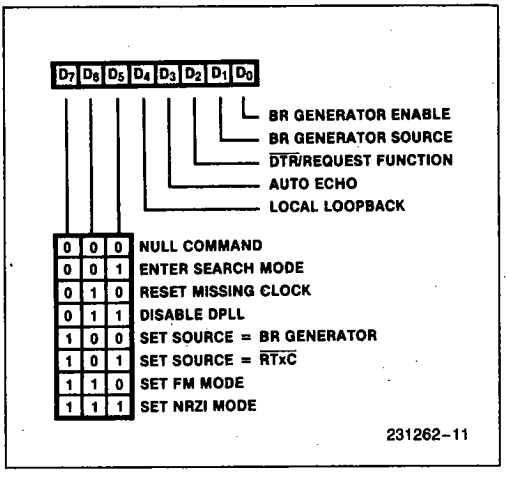


Figure 2. Write Register 14

WR14 / bit D1 = 0 → Clock comes from pin RTxC

WR14 / bit D1 = 1 → Clock comes from System Clock (PCLK)

On RESET WR14 / bit D1 = 0.

It should be noted that for the case of Bit D1 = 0, the clock comes either from:

a. Clock on pin RTxC - if WR11 / D7 = 0

or b. Crystal on pins RTxC & SYNC - if WR11 / D7 = 1

Step 4: BRG Enable

This is the last step where bit D0 of WR14 is set to start the BRG. The BRG can also be disabled by resetting this bit.

APPENDIX B
MODEM CONTROL PINS ON THE 82530

Introduction

This article describes how the CTS/ and CD/ pins on the 82530 behave and how to write software to service these pins. The article explains when the External Status Interrupt occurs and how and when to issue the Reset External Status Interrupt command to reliably determine the state of these pins.

Bits D3 and D5 of register RR0 show the *inverted* state of logic levels on CD/ and CTS/ pins respectively. It is important to note that the register RR0 does *not* always reflect the *current* state of the CD/ and CTS/ pins. Whenever a Reset External Status Interrupt (RESI) command is issued, the (inverted) states of the CD/ and the CTS/ pins get updated and latched into the RR0 register and the register RR0 then reflect the inverted state of the CD/ and CTS/ pins at the time of the write operation to the chip. On channel or chip reset, the inverted state of CD/ and CTS/ pins get latched into RR0 register.

Normally, a transition on any of the pins does not necessarily change the corresponding bit(s) in RR0. In certain situations it does and in some cases it does not. A sure way of knowing the current state of the pins is to read the register RR0 *after* a RESI command.

There are two cases:

I. External Status Interrupt (ESI) enabled.

II. Polling (ESI disabled).

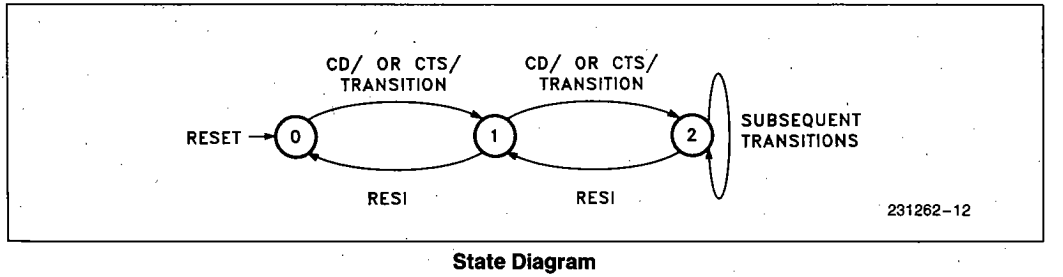
Case I: External Status Interrupt (ESI) Enabled

Whenever ESI is enabled, an interrupt can occur whenever there is a transition on CD/ or CTS/ pins - the IE bits for CD/ and/or CTS/ must also be set in WR15 for the interrupt to be enabled.

In this case, the first transition on any of these pins will cause an interrupt to occur and the corresponding bit in RR0 to change (even without the RESI command). A RESI command resets the interrupt line and also latches in the current state of both the CD/ and the CTS/ pins. If there was just one transition the RESI does not really change the contents of RR0.

If there are more than one transitions, either on the same pin or one each on both pins or multiple on both pins, the interrupt would get activated on the first transition and stay active. The bit in RR0 corresponding only to the very first transition is changed. All subsequent transitions have no effect on RR0. The first transition, in effect, freezes all changes in RR0. The first RESI command, as could be expected, latches the final (inverted) state of the CD/ and CTS/ pins into the RR0 register. Note that all the intermediate transitions on the pins are lost (because the response to the interrupt was not fast enough). The interrupt line gets reset for only a brief moment following the first RESI command. This brief moment is approximately 500 ns for the 82530. After that the interrupt becomes active again. A second RESI command is necessary to reset the interrupt. Two RESI commands resets the interrupt line independent of the number of transitions occurred.

Whenever operating with ESI enabled, it is recommendable to issue two back-to-back RESI commands and then read the RR0 register to reliably determine the state of the CD/ and CTS/ pins and also to reset the interrupt line in case multiple transitions may have occurred.



State Diagram

Case II: Polling RR0 for CD/ and CTS/ Pins

If RR0 is polled for determining the state of the CD/ and CTS/ pins, then the External Status Interrupt (ESI) is kept disabled. In this case the bits in RR0 may not change even for the first transition. The best way to handle this case to always issue a RESI command before reading in the RR0 register to determine the state of CD/ and CTS/ pins. Note, however, if two back-to-back RESI commands were to be issued every time before reading in the RR0 register, the first subsequent transition will change the corresponding bit in RR0.

The state diagram above illustrates how each transition on CD/ and CTS/ pins affect the 82530 and what effect the RESI command has.

State 0

It is entered on reset. No ESI due to CTS/ or CD/ are pending in this state. Any transition on CTS/ or CD/ pins lead to the state 1 *accompanied by an immediate change in the RR0 register.*

State 1

Interrupt is active (if enabled). If a RESI command is issued, state 0 is reached where interrupt is again inactive. However, a further transition on CTS/ or CD/ pin leads to state 2 *without an immediate change in RR0 register.*

State 2

Interrupt is active (if enabled). Any further transitions have no effect. A RESI command leads to state 1, temporarily making the interrupt inactive.

CONCLUSIONS

Register RR0 does not always reflect the current (inverted) state of the CD/ and CTS/ pins. The most reliable way to determine the state of the pins in interrupt or polling mode is to issue two back-to-back RESI commands and then read RR0. While polling, the second RESI is redundant but harmless. When issuing the back-to-back RESI commands to 82530 note that the separation between the two write cycles should be at least 6 CLK + 200 ns; otherwise the second RESI will be ignored.

APPENDIX C.
Interfacing 82530 to 80186

INTRODUCTION

The 82530 is Intel's new sophisticated dual channel multiprotocol serial communications controller. It can run up to 1.5 Mb/s in synchronous mode. It has useful features like on-chip baud rate generators and oscillators. It can be operated in polled, interrupt, half-duplex DMA, or full-duplex DMA modes. It is also capable of supplying its own interrupt vector during INTA cycles (like the 8274).

Interfacing the 82530 to the 8086/88 and 80186/188 processors requires the external logic shown in Figure 1.

FOUR TTL PACKAGE INTERFACE

A method of interfacing the 82530 to the 80186 CPU with four 14-pin TTL packages is described in this application note. The circuitry is shown in Figure 2. The TTLs are 74LS04, 74LS74, and 74LS08.

The interface supports the following operational modes:

1) Polled
2) Interrupt in vectored mode
3) Interrupt in non-vectored mode
4) Half-duplex - DMA on both channels
5) Full-duplex - DMA on one channel

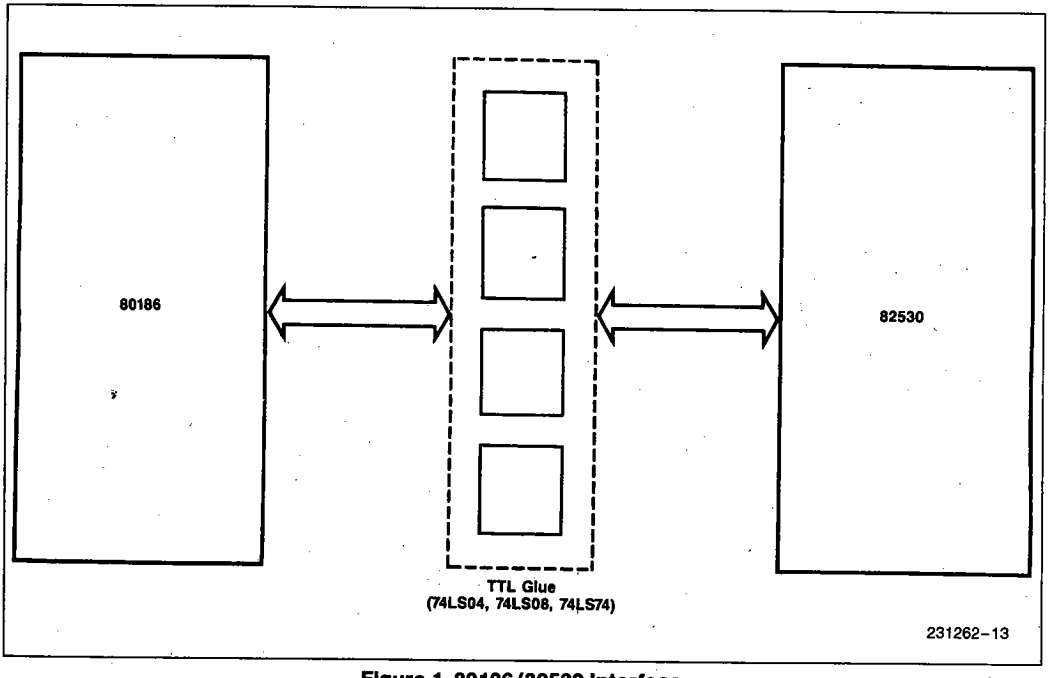


Figure 1. 80186/82530 Interface

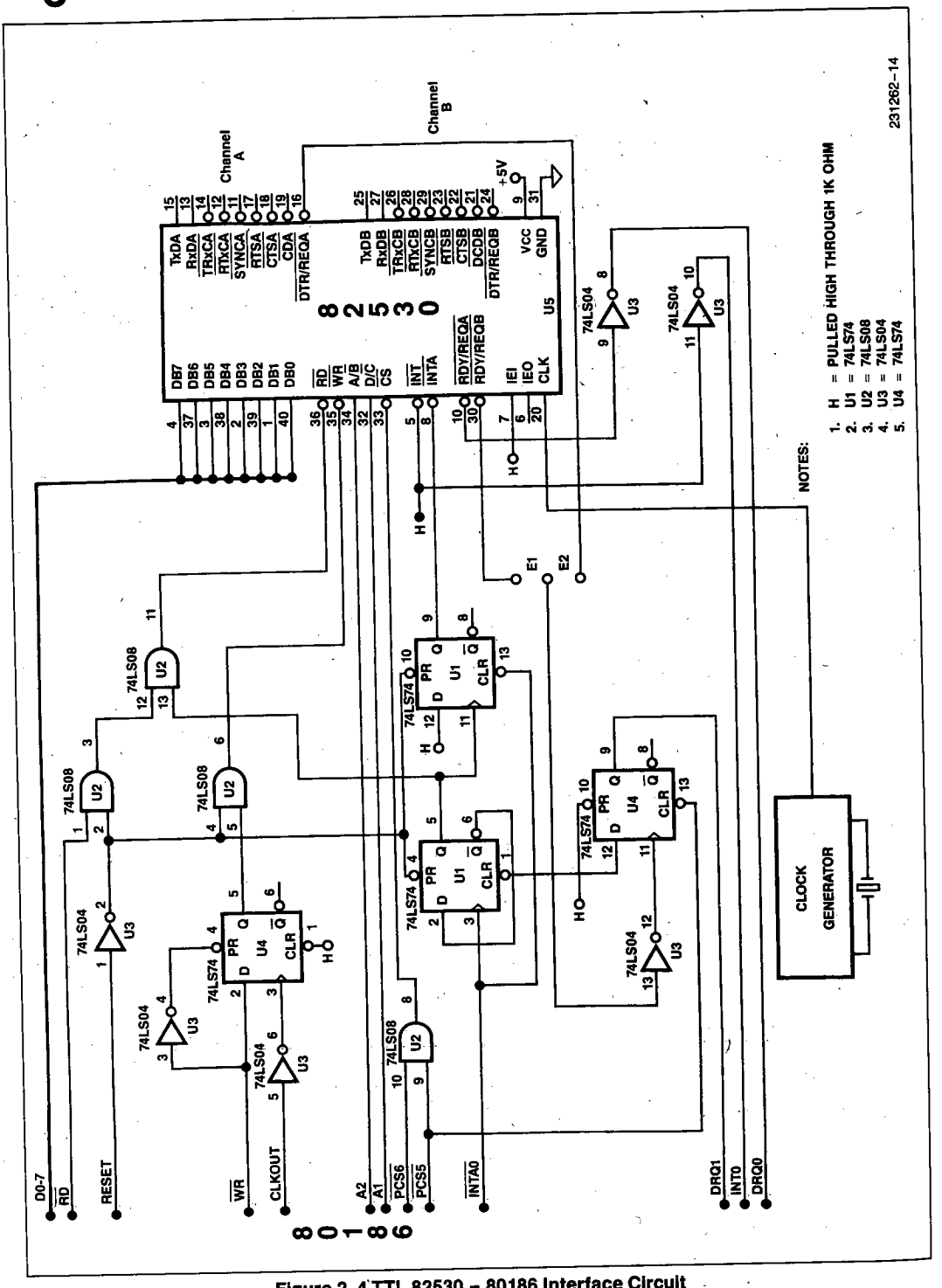


Figure 2. 4 TTL 82530 – 80186 Interface Circuit

231262-001

PRINCIPLES AND CIRCUIT DESCRIPTION

The principles shown can be used easily to extend full duplex DMA to both channels. This can in fact be done using the same 4 TTL packages if an 8288 were also used in the system—more of that later. The reason why TTL interfacing is necessary and how it is done is now described.

A) **Reset**

The 82530 does not have an explicit hardware reset input; however, simultaneous activation of $\overline{RD}$ and $\overline{WR}$ signals, as shown in Figure 3, is equivalent to a hardware reset of the 82530. This requires ORing of $\overline{RESET}$ with $\overline{RD}$ and $\overline{WR}$ signals to the 82530.

B) **Write**

The falling edge of $\overline{WR}$ should not occur before the data (to be written to the 82530) is valid (see Figure 4). Nor should the rising edge of $\overline{WR}$ occur after the data becomes invalid. This means that the $\overline{WR}$ active phase should occur entirely during the time when the data is valid. The $\overline{WR}$ signal from 8086/88/186/188 goes active before the data is valid. A D flip-flop and two inverters are used to delay the $\overline{WR}$ going to 82530 so that it becomes active after

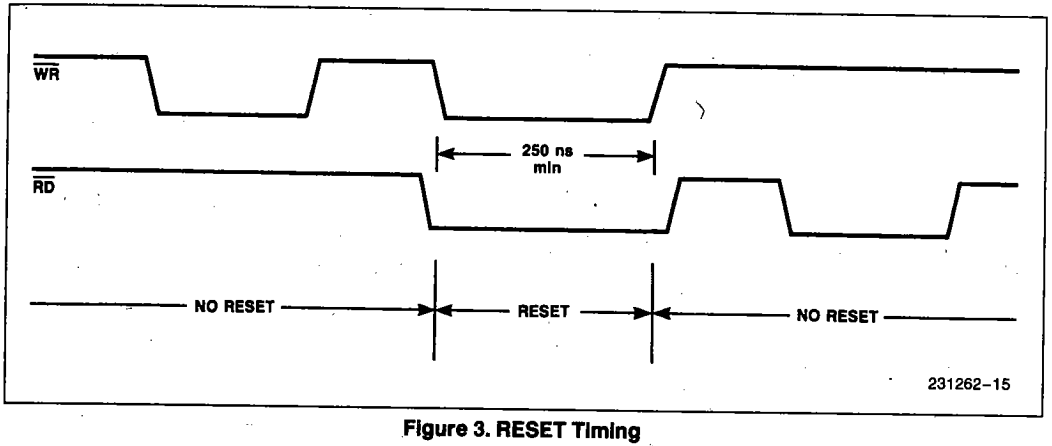


Figure 3. RESET Timing

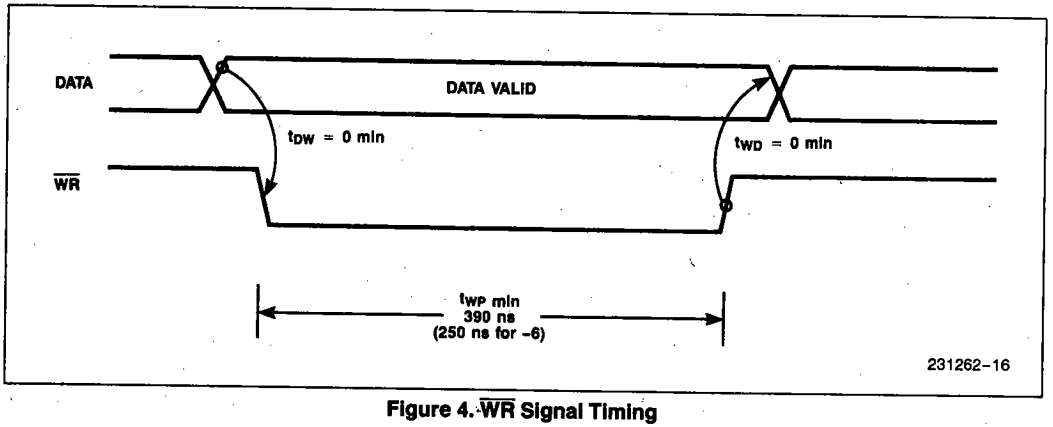


Figure 4. $\overline{WR}$ Signal Timing

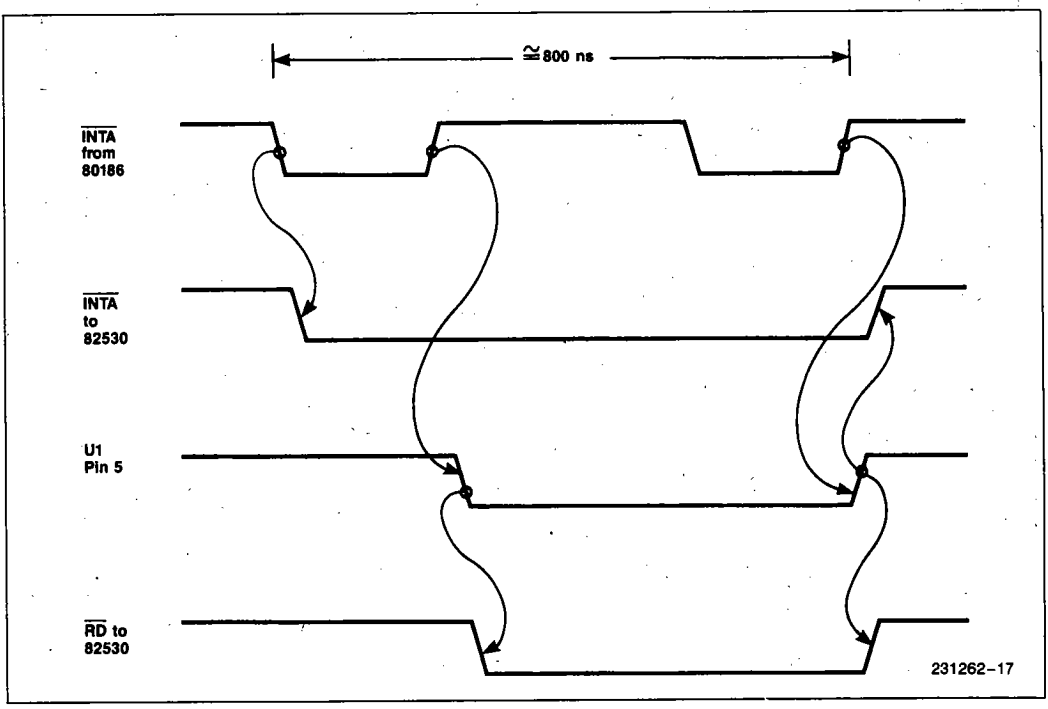


Figure 5. INTA Signal Processing

the data is valid. Note that if an 8288 is used to generate the IOWR signal (as in all big systems), then the flip-flop and inverters are not required since IOWR from the 8288 is compatible with the 82530 timing requirements.

C) DMA

The 82530 has two types of DMA request outputs; also, it has no DACK inputs. This means that the 82530 requires either a two cycle type of DMA transfer (a la 80186/88 or 8089), or DACK from the DMA controller (e.g. 8237A) has to be used to generate CS, A/B, and D/C signals.

The first type of DMA request is RDY/REQ. It can be programmed to function as RDY or DMAREQ (WR1: Bit 6). It can further be programmed as DMAREQ for transmit or for receive (WR1: Bit 5). This enables using just one signal for both the receive and transmit functions—ideal for half-duplex operation. This signal needs just an inversion to be fed into the DRQ input of the 80186.

The second DMA request signal is DTR/REQ. It can be programmed to function as DTR (Data Terminal Ready) or as DMAREQ for transmitter (active on transmitter buffer empty) in WR14: Bit 2. Thus, full-duplex DMA is possible by using DTR/REQ as TxDRQ and RDY/REQ as RxDRQ. DTR/REQ requires a little over 5 CLK cycles to become inactive. This would cause the DMA controller to run multiple DMA cycles, causing loss of data. A flip-flop is set by DTR/REQ whose output is DRQ1 to the 80186. The response of the 80186 to DRQ1 is a read or write at PCS5 address to do the DMA TRANSFER. This resets the flip-flop cutting off the DMA request to the 80186 which prevents false DMA transfer.

The DMA configurations supported by the interface are:

- Half-duplex on Channel A and Channel B
- Full-duplex on Channel A and no DMA on Channel B

D) INTA Processing

80186 generates 2 back-to-back INTA cycles in response to an interrupt and expects to read the interrupt vector on the second cycle. Two flip-flops (U1) are used to convert these two cycles to one INTA cycle and a RD pulse as required by the SCC. See timing diagram in Figure 5. SCC requires that the RD pulse is contained within the INTA pulse. This, along with the pulse width requirements for INTA and RD signals are easily met.

WAIT STATE REQUIREMENTS

The 82530 requires wait states in a normal single buffered system, as shown in Figure 6. They arise primarily due to the WR pulse width (=390 ns) and its timing with respect to data valid as shown in Figure 4.

		SCC	
		82530 (4 MHz)	82530-6 (6 MHz)
Processor	80186-6 (6 MHz)	2	1
	80186 (8 MHz)	3	2

Figure 6. Wait State Requirements

It is assumed in this interface design that the 80186 generates the chip selects and the appropriate number of wait states. In an 8086/88 system, chip select and wait states must be generated externally just as for all other peripheral components attached to the CPU.

The $\overline{PCS6}$ chip select output from the 80186 is used to select the 82530 for all operations except to service DMA on Channel 1 of the 80186 when $\overline{PCS5}$ is used. Note that it is necessary to pulse $\overline{PCS5}$ signal before enabling the DMA Channel 1. This resets the DRQ1 flip-flop. A block for clock generator is also shown—although it is not considered a part of the CPU interface. It may be easily derived from CLKOUT.

The 4 TTL pack interface presented here covers all features of the SCC usage. In many cases the interface need not be as extensive as shown here and results in saving board space. Two cases where considerable saving is achieved are:

Case 1: System Using 8288

If the system uses an 8288 bus controller for 80186, pre-processing of $\overline{WR}$ input is not necessary and the $\overline{IOWC}$ output of 8288 can be fed directly to pin 5 of U2 (74LS08). This is because $\overline{IOWC}$ signal meets the timing requirements of the SCC. Also note, that the interface circuit is then totally independent of the 80186 clock.

Case 2: System Using Non-Vectored Interrupt Mode for SCC

Such a system will not need the component U1 (74LS74) nor the AND gate U2 (pins 11, 12, 13). Pin 3 of U2 can be fed directly to the $\overline{RD}$ input of SCC.

CONCLUSION

This four TTL package interface solution is low cost and compact (1.2 sq. inch). It should satisfy 82530 interfacing for almost all applications. In fact, as already mentioned, many applications may require only 2–3 TTL packages for interfacing the 82530 to 80186 or to other INTEL processors.